0.1. Training Configuration.

To demonstrate the efficiency of on-device training, we use Nvidia Jetson Nano GPU with 4-GB memory as our training platform. We evaluate the model performance using PyTorch as the simulation platform. Note that, the reported activation memory usage is calculated by our definition in Table 1, since PyTorch does not support explicit fine-grained memory management. In the training, for ResNet-50, we use Adam as the optimizer with cosine learning rate decay, an initial rate of 1e-3, and the number of iteration was set to 30. For ResNet-26 training on the challenge dataset, we use an SGD optimizer with an initial learning rate of 0.1. We schedule the learning rate decay at 40, 80 and 100 epoch with a rate of 0.1. Again, as shown in Fig. 1, we use right configuration of DA for ResNet-50 and left configuration to train ResNet-26 model.

Figure 1. Illustration of integrating the proposed DA in popular basic block and bottleneck block in ResNets. Note, black indicates pre-trained backbone model, blue indicates added modules.

0.2. Learning the dynamic spatial gate

First, we adopt a continuous logistic function:

\[ \sigma(G(H_s(A))) = \frac{1}{1 + \exp(-\beta G(H_s(A)))}, \quad (1) \]

where \( \beta \) is a constant scaling factor. Note that the logistic function becomes closer to the hard thresholding function for higher \( \beta \) values.

Then, to learn the binary mask, we leverage the Gumbel-Sigmoid trick, inspired by Gumbel-Softmax [1] that performs a differential sampling to approximate a categorical random variable. Since sigmoid can be viewed as a special two-class case of softmax, we define \( p(\cdot) \) using the Gumbel-Sigmoid trick as:

\[ p(G(H_s(A))) = \frac{\exp((\log \pi_0 + g_0)/T)}{\exp((\log \pi_0 + g_0)/T) + \exp((g_1)/T)}, \quad (2) \]

where \( \pi_0 = \sigma(m^r) \). \( g_0 \) and \( g_1 \) are samples from Gumbel distribution. The temperature \( T \) is a hyperparameter to adjust the range of input values, where choosing a larger value could avoid gradient vanishing during back-propagation. Note that the output of \( G(H_s(A)) \) becomes closer to a Bernoulli sample as \( T \) is closer to 0. We can further simplify Eq. 2 as:

\[ p(G(H_s(A))) = \frac{1}{1 + \exp(-(\log \pi_0 + g_0 - g_1)/T)} \quad (3) \]

Benefiting from the differential property of Eq. 1 and Eq. Eq. (3), the real-value mask \( m^r \) can be embedded with existing gradient based back-propagation training. To represent \( p(G(H_s(A))) \) as binary format \( G^b \), we use a hard threshold (i.e., 0.5) during forward-propagation of training. Because most values in the distribution of \( p(m^r) \) will move towards either 0 or 1 during training, generating the binary mask by \( p(G(H_s(A))) \) could have more accurate decision, resulting in better accuracy.

\[^1\text{https://forums.developer.nvidia.com/t/pytorch-for-jetson-version-1-7-0-now-available/72048} \]